Machine_Learning_Project

March 18, 2025

MA124 Maths by Computer

1 Project: Constructing and applying Machine Learning models

Background Machine Learning is the use and development of computer systems that are able to learn and adapt without following explicit instructions, by using algorithms and statistical models to analyse and draw inferences from patterns in data. In machine learning we are interested in creating functions or models which fit real data. Such functions can be found by making them dependent on a number of parameters which are updated over and over again to improve the fit to the data. You will explore in this project.

Suggested reading There are many great articles, books, videos about machine learning. This list just picks out a few:

Linear Regression Model. Short, simple article covering a task similar to A2 below.

Logistic Regression Model An article which covers models similar to those in A3 and A4 below.

What is a neural network. Great YouTube series of introductory videos about training neural networks by the excellent 3Blue1Brown, well worth watching.

Blog: An Introduction to Neural Networks. Good discussion of the basic ideas used in this project.

Hands on Machine Learning with Python This might accompany a full course on machine learning. It has much more detail that you will need. As the title says, it's all done using Python. It's well-written and popular.

Struture of project There are 6 tasks in the document below, tasks A1-A4 in Section A and then tasks B1 and B2 in section B. Your group should do all the tasks in section A and at least one of the two tasks in section B. A1-A4 are worth approximately 60% of the credit for this submission and section B is worth approximately the other 40%.

Notes on submission Read through the document **MA124 Maths by Computer Tutor Group Projects Information for Students.pdf** on the MA124 Moodle page.

Before submitting see the notes at the end of this document.

Allowed libraries for this project: numpy matplotlib copy pandas scikitlearn. **Note you may not use scikitlearn in any of tasks A2, A3 or A4.**

This means that you may include these lines of code at the start of of your code cells (and please use the aliases given below).

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import copy
import sklearn # Not to be used in any of tasks A2, A3 or A4.
```

1.1 Section A - (worth approximately 60% of the marks)

1.1.1 Task A1 - Linear Regression machine learning task using SciKitLearn (worth approximately 20% of the marks)

In a recent research article published in the journal Computer Communications, authors Sathishkumar V E, Jangwoo Park, and Yongyun Cho sought to predict the "bike count required at each hour for the stable supply of rental bikes"[1]. They employed several regression models, including linear regression. The dataset used in the original study is available here.

Task: Apply machine learning to a modified version of the original dataset and report the results.

[1] Sathishkumar V E, Jangwoo Park, and Yongyun Cho. 'Using data mining techniques for bike sharing demand prediction in metropolitan city.' Computer Communications, Vol.153, pp.353-366, March, 2020 web link.

The original research article and a modified dataset are posted on the MA124 Moodle page in the resources for this project. You will need to refer to the article for some of the tasks below. You will need to download SeoulBikeData_mod.csv and put it into the folder with this file.

SeoulBikeData_mod.csv has been modified from the original dataset to remove the categorical variables, and to convert dates to months. Months have been coded by number, e.g. 1 = January, etc. Only half the months are included in the modified dataset.

While the number of tasks is large, this is in part because the instructions are rather specific. Many of them are very similar to things you saw in term 1, in the week 8 and week 10 notebooks and lectures.

Specifically, your code should

1. Import needed libraries. (You will need pandas, as well as things from sklearn, and of course numpy and matplotlib.)
2. Using pandas, read SeoulBikeData_mod.csv into a Dataframe.
3. `describe` the Dataframe.
4. Plot a histogram of `Rented Bike Count`. Do not plot this as a density, but as a count. See Fig. 3 of the article. The vertical axis in the article is labelled "frequency", but is the same as the count.

   Produce a box plot similar to that in Fig. 3 of the article.

Try to generate both the histogram and box plot to look approximately as they do in the article.

5. From the full Dataframe, create a new Dataframe `X` containing all the columns except `Rented Bike Count` and a Series `y` containing only the `Rented Bike Count` column. These are your design matrix and target respectively.
6. Perform a test-train split to create `X_train`, `X_test`, `y_train` and `y_test`. You **must** use the same percentage of data for testing and training as was used in the article and you **must** state what they are. You can find these in the article.
7. Create and train a linear regression model.
8. Use the trained model to obtain `y_pred`, the prediction on the test data `X_test`. Form the residual `resid = y_test - y_pred`.
9. Compute and report: Rsquared (R2), the Root Mean Squared Error (RMSE), the Mean Absolute Error (MAE), and the Coefficient of Variation (CV). Compare these results to those on the top, right of Table 4 of the article. (Note, the modified dataset we are studying is different from that used in the article. Hence the results will not be identical. However, the procedure is very close to that used in the article.)
10. Produce and comment on the following plots.

- Histograms of `y_test` and of `y_pred` (on the same plot). These should be reported as counts rather than densities.
- A scatter plot of `resid` as a function of `y_test` corresponding to Fig. 9 of the article. (Recall what `y_test` represents and label the plot appropriately.) Unlike Fig. 9 of the paper, you should use a colormap to plot the different `Hours` in different colours.
- A scatter plot of `resid` as a function of `X_test['Month']`. Use a colormap to indicate the absolute value of `resid`.
- A scatter plot of `resid` as a function of `X_test['Rainfall(mm)']`. Use a colormap to indicate the absolute value of `resid`.

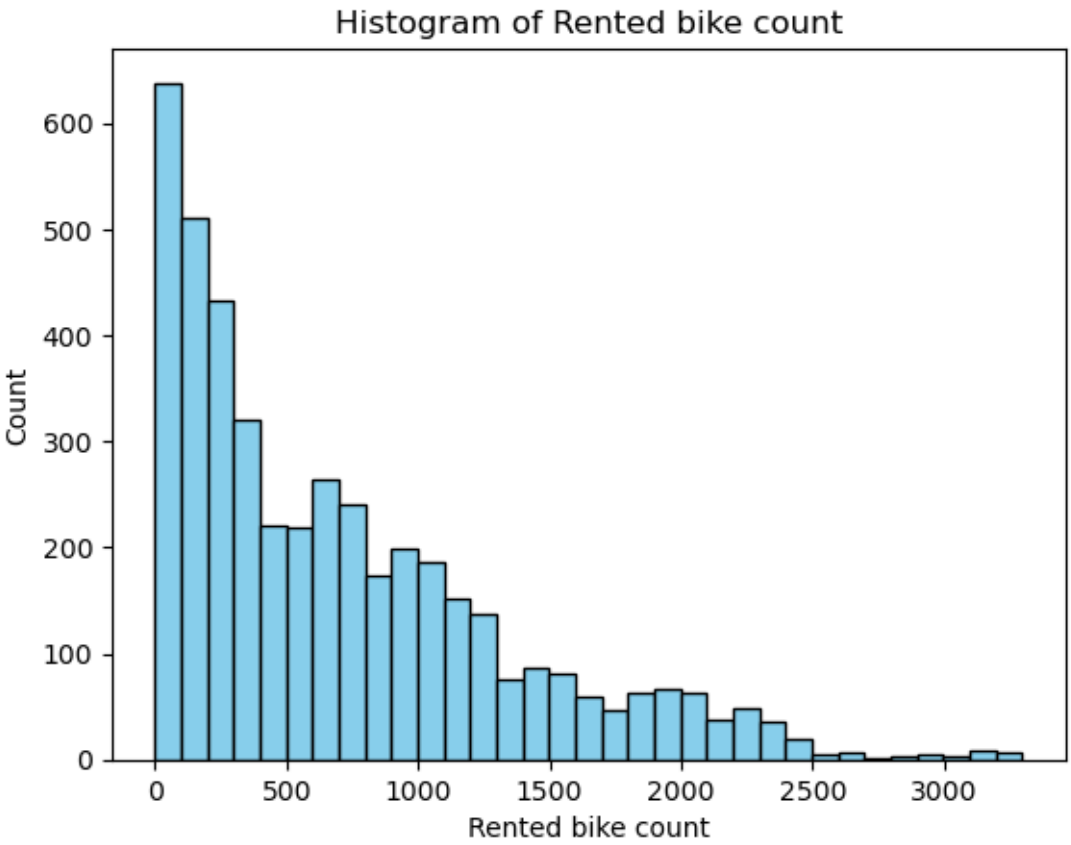
Insert code and markdown cells below in which to answer this.

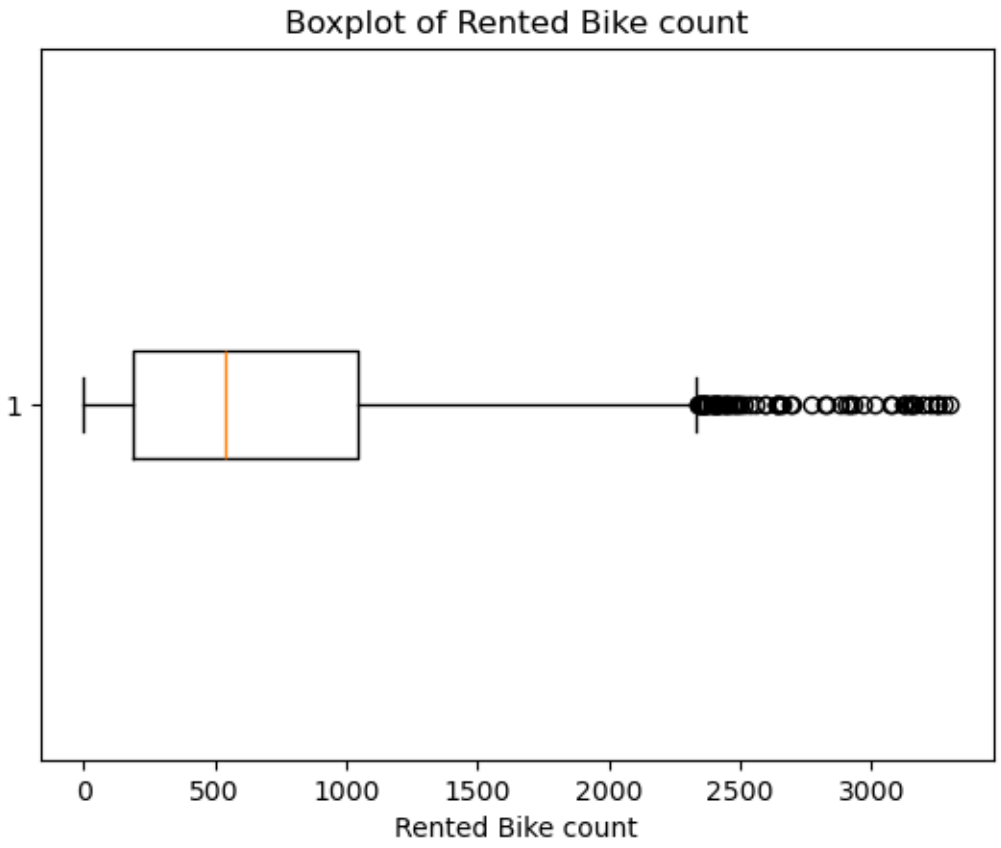
[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sklearn
from sklearn import linear_model


seoulbike = pd.read_csv("SeoulBikeData_mod.csv")
seoulbike.describe()
```

[2]:
```
       Rented Bike Count         Hour  Temperature(°C)  Humidity(%)  \
count        4416.000000  4416.00000      4416.000000  4416.000000
mean          704.591259    11.50000        13.203986    59.257246
std           637.407244     6.92297        11.529724    20.162981
```

```
min             0.000000     0.00000       -17.800000     0.000000
25%           189.000000     5.75000         4.500000    44.000000
50%           542.000000    11.50000        14.300000    58.000000
75%          1047.000000    17.25000        22.200000    75.000000
max          3298.000000    23.00000        38.000000    98.000000

       Wind speed (m/s)  Visibility (10m)  Dew point temperature(°C)  \
count       4416.000000       4416.000000                4416.000000
mean           1.653193       1398.641304                   4.723958
std            0.976361        621.560536                  12.671445
min            0.000000         27.000000                 -30.600000
25%            0.900000        853.000000                  -3.500000
50%            1.500000       1621.000000                   6.400000
75%            2.200000       1999.000000                  14.900000
max            6.700000       2000.000000                  26.800000

       Solar Radiation (MJ/m2)  Rainfall(mm)  Snowfall (cm)        Month
count              4416.000000   4416.000000    4416.000000  4416.000000
mean                  0.560181      0.139312       0.070879     5.956522
std                   0.860496      1.145271       0.437416     3.407262
min                   0.000000      0.000000       0.000000     1.000000
25%                   0.000000      0.000000       0.000000     3.000000
50%                   0.010000      0.000000       0.000000     5.000000
75%                   0.910000      0.000000       0.000000     9.000000
max                   3.520000     35.000000       8.800000    11.000000
```

[3]:
```
y = seoulbike['Rented Bike Count']
plt.hist(y,bins=33,color='skyblue', edgecolor='black')
plt.xlabel("Rented bike count")
plt.ylabel("Count")
plt.title("Histogram of Rented bike count")
plt.show()

plt.boxplot(y,vert=0)
plt.xlabel("Rented Bike count")
plt.title("Boxplot of Rented Bike count")
plt.show()
```


Histogram of Rented bike count
Count
Rented bike count
0
100
200
300
400
500
600
0
500
1000
1500
2000
2500
3000



75% of the final data is utilized for model training and the remaining 25% of the data is used for testing purpose.

[4]:
```
X=seoulbike.drop(['Rented Bike Count'], axis=1)
y=seoulbike['Rented Bike Count']

X_train=X.iloc[:-1104,:]
X_test=X.iloc[-1104:,:]
y_train=y[:-1104]
y_test=y[-1104:]


regr = linear_model.LinearRegression()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)
resid = y_test - y_pred
```

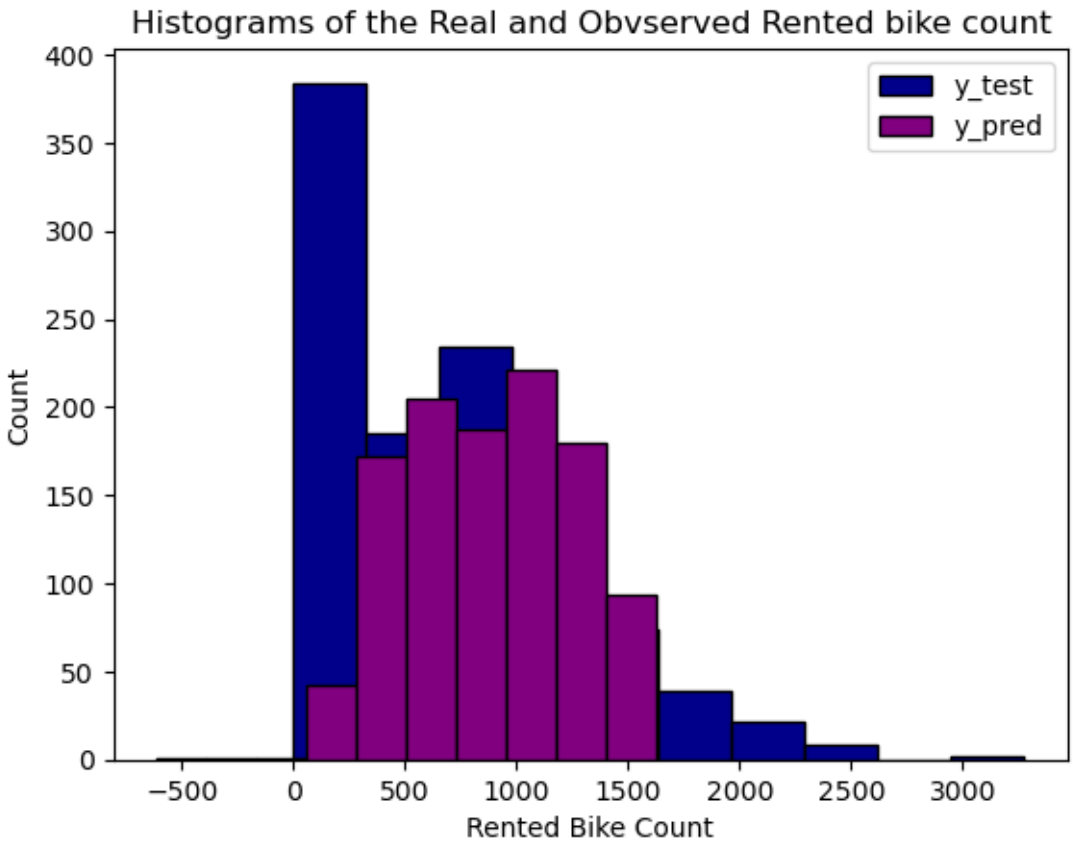
[5]:
```
resid2= (y_test - y_pred)**2
square = (y_test - np.mean(y_test))**2
R2 = 1 - (sum(resid2)/sum(square))
```

```
print("the value of R2 is", R2)
RMSE = np.sqrt((sum(resid2)/1104))
print("the RMSE value is",RMSE)
MAE = sum(abs(resid))/1104
print("the MAE value is",MAE)
CV = (np.sqrt(sum(resid2)/1104))/np.mean(y_test)*100
print("the CV value is", CV)
```

```
the value of R2 is 0.0936546799282616
the RMSE value is 546.1983395427221
the MAE value is 404.5364341012174
the CV value is 80.92576565129693
```

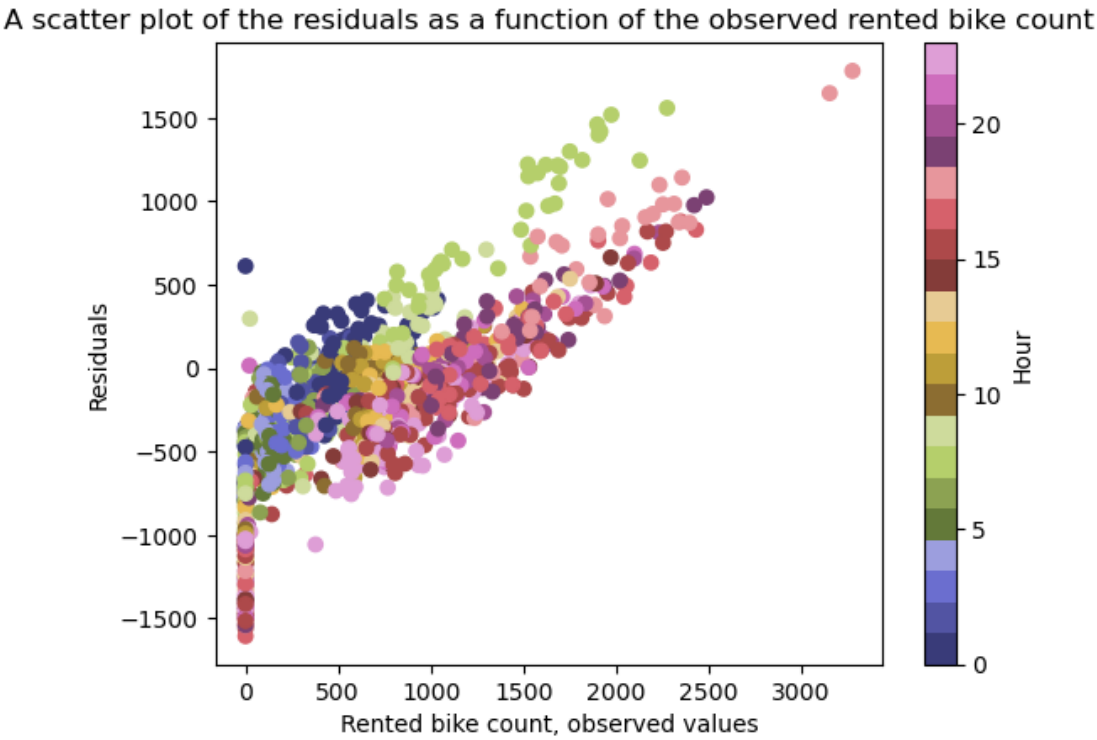
Values in the article: R2: 0.55 RMSE: 427.71 MAE: 322.32 CV: 61.03 The RMSE and MAE values are similar but larger than they are in the article showing greater error in this model. The CV and R squared values are more different to the article which could show that there is a difference in the mean of the test data in the article and this model. The R squared value is particularly different and low which does not show that this model is very accurate. This could be because the dataset is modified from the article and only contains half the months. The model would also likely be more accurate if I had not taken the first 75% of the data for training but taken the training data from throughout the dataset so it was more representative of the whole year and not just a few months.

[6]:
```
plt.hist(y_test,color='darkblue',edgecolor='black', label="y_test")
plt.hist(y_pred,color='purple',edgecolor='black',label="y_pred")
plt.xlabel("Rented Bike Count")
plt.ylabel("Count")
plt.title("Histograms of the Real and Obvserved Rented bike count")
plt.legend()
plt.show()
```



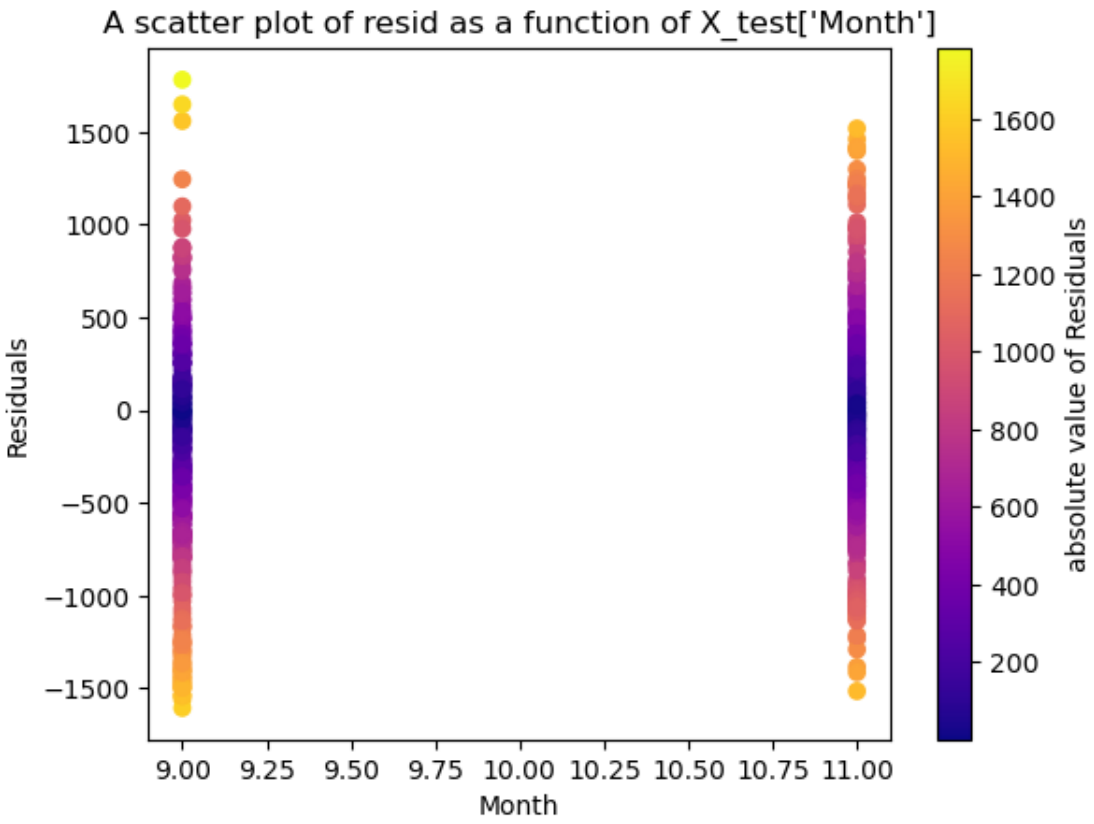
The model seems to fit relatively well from when the rented bike count value is 250 to just over 1500 but it is very different in other areas. The model appears to predict a negative rented bike count which is not possible and therefore not a useful prediction. The initial histogram of the whole dataset shows that the data is skewed to the right as it has a long right tail and the histogram for the test data is similar but the shape of the predicted values is completely different. At zero the count should be the highest but instead the model does not predict it to happen more than 50 times when in it should be close to 400 which is a large issue in the model. It shows that the model is overpredicting at these hours. This histogram also shows that the model does not predict anything above the count 1500 and while some of these values would be classed as outliers as seen in the inital boxplot this is not the case for all of them and shows a large issue in the model.

[7]:
```
Hours=seoulbike['Hour']
plt.scatter(y_test,resid,c=Hours[-1104:], cmap='tab20b')
plt.colorbar(label="Hour")
plt.ylabel('Residuals')
plt.xlabel('Rented bike count, observed values')
plt.title("A scatter plot of the residuals as a function of the observed rented␣
 ↪bike count")
plt.show()
```



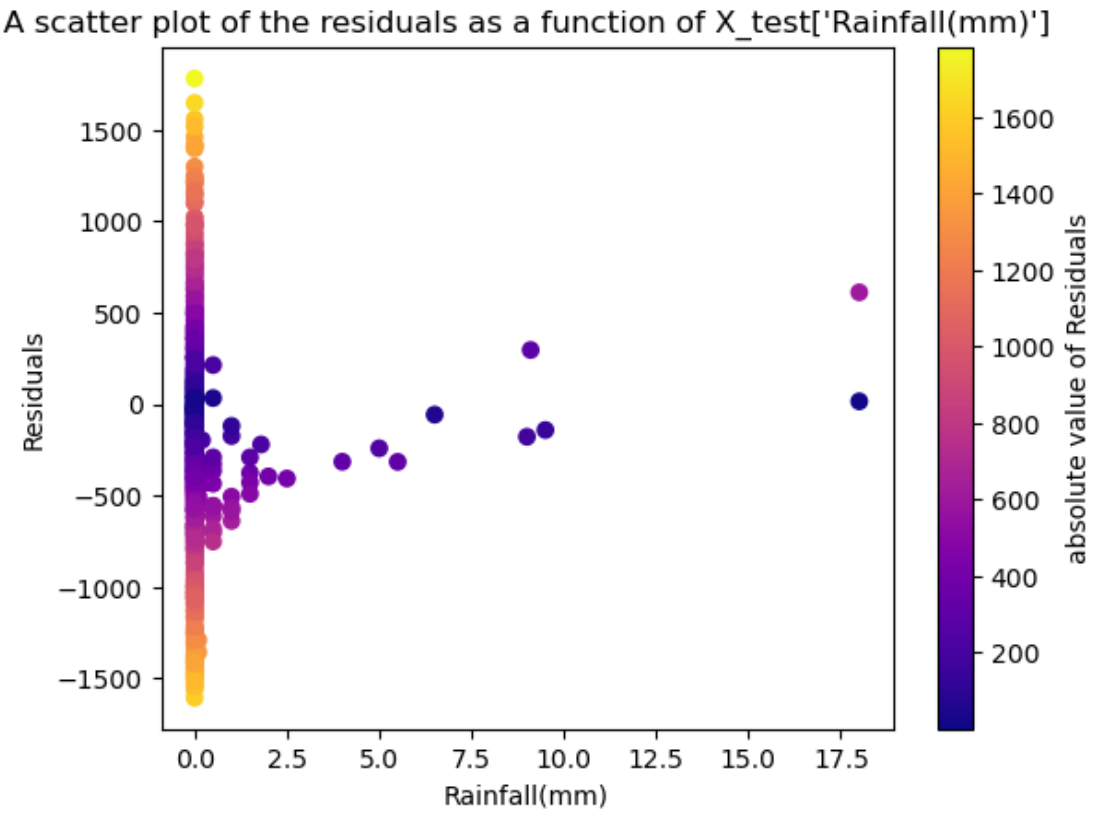
The residuals are negative except in two cases at 0 which shows that the model frequently predicts the count to be positive when it is zero. In general the residuals seem to increase as the rented bike count increases showing a greater underprediction for the more extreme counts. The colourmap shows that the highest rented bike count values happen in the same 2 hours in the morning and the evening which the value of the residulas being greater in the morning so the model appears to have less accurate predictions then. For the rest of the data the residuals seem to be within the range of -750 to 500.

[8]:
```
plt.scatter(X_test['Month'],resid,c=abs(resid),cmap="plasma")
plt.ylabel('Residuals')
plt.xlabel('Month')
plt.colorbar(label="absolute value of Residuals")
plt.title("A scatter plot of resid as a function of X_test['Month']")
plt.show()
```



x_test only contains data from two months since I have taken the first 75% percent of the data for training and it is ordered by month. The absolute value of the residuals reaches a higher value in september than in november which means that there are hours where the error of the prediction is larger. Overall the residuals take similar values in both months but since none of the training data is from either of these months the model is not able to see patterns in the data for these months to use that as a factor for the predictions.

[9]:
```
plt.scatter(X_test['Rainfall(mm)'],resid,c=abs(resid), cmap="plasma")
plt.ylabel('Residuals')
plt.xlabel('Rainfall(mm)')
plt.title("A scatter plot of the residuals as a function of␣
 ↪X_test['Rainfall(mm)']")
plt.colorbar(label="absolute value of Residuals")
plt.show()
```



Most of the data lies when there is no rainfall but for the pointes where there is rainfall the absolute value of the residuals does not reach above around 600 which is small compared to other points. Comparitively the model is able to predict the rented bike count better when there is rainfall to when there is not. There is a large range in the absolute value of the residuals when there is no rainfall, since this is the case most of the time there could be other factors affecting the prediction other than the lack of rainfall.

1.1.2 Task A2 - Linear Regression, on a very small dataset, from scratch with only numpy (worth approximately 10% of the marks)

(illustrating idea of rates of change with respect to the model parameters and gradient descent) Consider a really small dataset consisting of only three points in the plane: $(1, 1), (2, 4), (3, 5)$.

For each point (training example) consider the x-coordinate as a single input feature, and the y-coordinate as the output or label of the example. The objective is to find the best line to fit these data. Here 'best' will mean the line which minimises the mean of the sum of the squares of the residuals with respect to the three points.

In this task and in the following tasks A3 and A4 we will use a slightly different convention to store our data with respect to the one used in A1 or the lectures. X_{train} and y_{train} will now be the transpose of the corresponding matrices in A1. In A3 and A4 this will be a more convenient form for applying functions given by matrices to the data with the matrix on the left of the data matrix.

The feature matrix X_{train} contains the feature vectors for all the training examples. More precisely $X_{\text{train}} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ has one column for each training example and each column is the feature vector for that training example (in this case we have only one feature which is the x-coordinate).

The labels matrix y_{train} contains the labels for all the training examples. More precisely $y_{\text{train}} = \begin{pmatrix} 1 & 4 & 5 \end{pmatrix}$ has one column for each training example and each column contains the label for that training example (the y-coordinate).

The model here is a simple linear model

$$\hat{y} = mx + c$$

with trainable parameters m and c. The goal is to find the optimal values of m and c to fit these training data. In the Machine Learning context this means defining a loss function between prediction and true label:

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

and a cost function which is simply the average on all training examples of these losses.

Specifically, you should

1. Show, in a markdown cell, how the cost J depends only on the trainable parameters m and c, and can be computed to be:

$$J = \frac{14m^2 + 12cm - 48m + 3c^2 - 20c + 42}{3}$$

2. Find, in a markdown cell, expressions for the partial derivatives $\dfrac{\partial J}{\partial m}$ and $\dfrac{\partial J}{\partial c}$.
3. Write a function `model1(alpha, num_iterations)` which takes as inputs a learning rate `alpha` and a number of iterations `num_iterations`, and returns optimized values of m and c through Gradient Descent. More precisely the function should initialize m and c randomly between -2 and 2 and then perform `num_iterations` steps of gradient descent with learning rate `alpha`. This means that the value of m is updated to $m - \dfrac{\partial J}{\partial m}\alpha$ and the value of c is updated to $c - \dfrac{\partial J}{\partial c}\alpha$ in each iteration. Your function should print the cost J periodically throughout the iteration process. You may wish to refer to the Machine learning project lecture for help with this. You may wish to build helper functions for the various tasks you need this function to do and refer to them.
4. Plot the 3 datapoints along with the optimal line $mx + c$ your model has found.
5. Explore the effect of changing the learning rate and the number of iterations.

Insert code and markdown cells here in which to answer this task

We have $\hat{y} = mx + c$ for the x coordinate of each point in the training data and $L(y, \hat{y}) = (\hat{y} - y)^2$ as well as the set of points $(1, 1)$, $(2, 4)$, $(3, 5)$. Substituting these into our equation for cost, a

simple geometric mean of the loss function evaluated at each of the three points, then expanding and siplifying:

$$\begin{aligned} J &= \frac{1}{3}\sum L(\hat{y}, y) \\ &= \frac{1}{3}(L(\hat{y}(1), 1) + L(\hat{y}(2), 4) + L(\hat{y}(3), 5)) \\ &= \frac{1}{3}((m + c - 1)^2 + (2m + c - 4)^2 + (3m + c - 5)^2) \\ &= \frac{1}{3}(m^2 + 2mc + c^2 - 2m - 2c + 1 + 4m^2 + 4mc + c^2 - 16m - 8c + 16 + 9m^2 + 6mc + c^2 - 30m - 10c + 25) \\ &= \frac{1}{3}(14m^2 + 12mc - 48m + 3c^2 - 20c + 42) \end{aligned} \tag{1}$$

So overall we have J dependant only on m and c , with

$$J = \frac{14m^2 + 12mc - 48m + 3c^2 - 20c + 42}{3} \tag{2}$$

The partial derivatives can then easily be found as

$$\frac{\partial J}{\partial m} = \frac{1}{3}(28m + 12c - 48) \tag{3}$$

$$\tag{4}$$

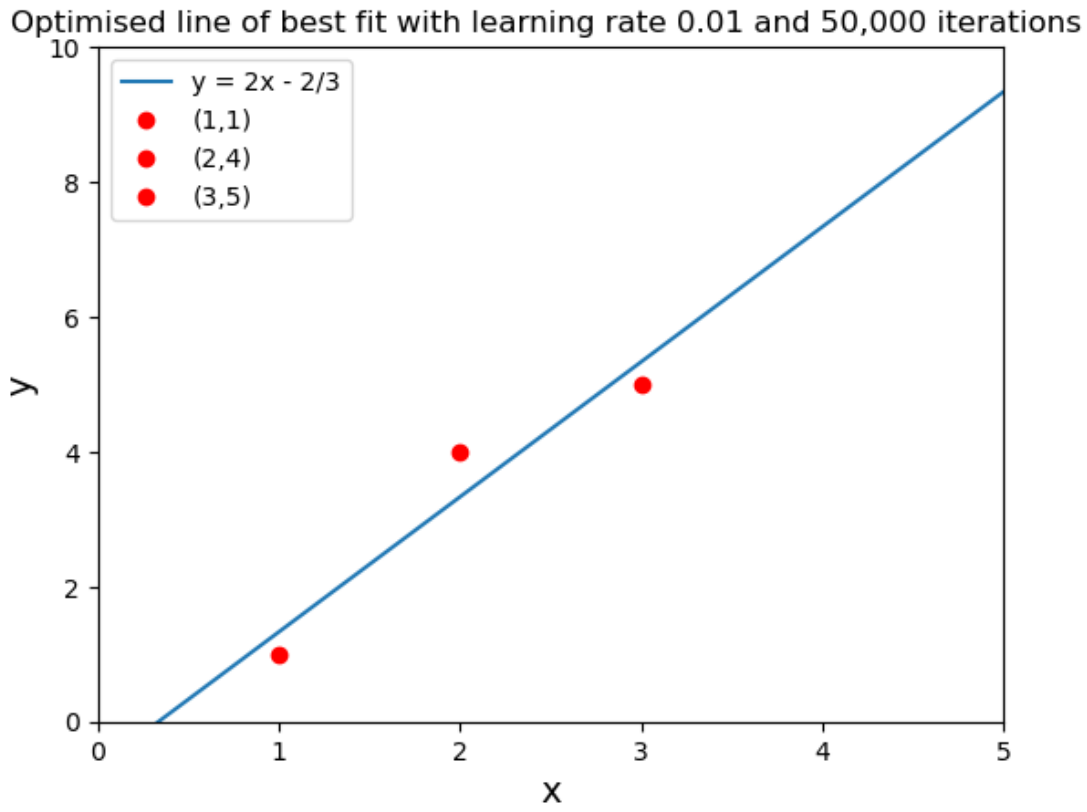
$$\frac{\partial J}{\partial c} = \frac{1}{3}(12m + 6c - 20) \tag{5}$$

[10]:
```
import numpy as np

## First, defining helper functions which take as inputs values of m and c and␣
 ↪return J and its partial derivatives evaluated at those values.

def J(m,c):
    return (1/3)*(14*m**2 + 12*c*m -48*m + 3*c**2 - 20*c +42)

def partialJwrtm(m,c):
    partialm = (1/3)*(28*m + 12*c -48)
    return(partialm)

def partialJwrtc(m,c):
    return (1/3)*(12*m + 6*c -20)

## Then, using these helper functions to define more helper functions which␣
 ↪take as inputs current values of m and c, as well as a learning rate alpha,␣
 ↪and return
## updated values of m and c, having undergone one step of gradient descent␣
 ↪using the specified learning rate.
```

```
def graddescentm(m,c,alpha):
    mnew = m - alpha*(partialJwrtm(m,c))
    return (mnew)

def graddescentc(m,c,alpha):
    cnew = c - alpha*(partialJwrtc(m,c))
    return (cnew)

## Finally, these helper functions are put together into the function for the␣
 ↪model, which takes as inuts a learning rate and number of iterations. The␣
 ↪model first
## randomises starting values of m and c, within the range specified in the␣
 ↪assignment, and then performs gradient descent with the learning rate the␣
 ↪number of
## times specified. The function also periodically prints the value of J, so␣
 ↪that it can be tracked and used to inform decisions about the number of␣
 ↪iterations used
## in the future. The output of the function is the final values of m and c.

def model1(alpha, num_iterations):
    m = np.random.uniform (-2,2)    ## randomising the starting values for m␣
 ↪and c
    c = np.random.uniform (-2,2)
    print("J_0 = ",J(m,c))
    for iterationnumber in range(1,num_iterations+1):
        m_updated = graddescentm(m,c,alpha)
        c_updated = graddescentc(m,c,alpha)            ## here, care is taken␣
 ↪to ensure that c doesn't update using the already updated value of m.
        m = m_updated                                  ## new values are␣
 ↪calculated for both m and c, and then they are updated to these values
        c = c_updated
        if iterationnumber%10000 == 0:
            print ("J_",iterationnumber,"=",J(m,c))
    return (m,c)
```

[11]:
```
import numpy as np
import matplotlib.pyplot as plt

##First, a new model is defined, one identical to the previous model except␣
 ↪that it doesn't print J. This makes it easier to extract our optimised␣
 ↪parameters.

def model2(alpha, num_iterations):
    iterationnumber = 0
    m = np.random.uniform (-2,2)
    c = np.random.uniform (-2,2)
```

```
    for iterationnumber in range(0,num_iterations):
        m_updated= graddescentm(m,c,alpha)
        c_updated= graddescentc(m,c,alpha)
        m = m_updated
        c = c_updated
    return (m,c)

m = model2(0.01,50000)[0]
c = model2(0.01,50000)[1]

x = np.linspace(0,10,10000)
y = m*x + c

plt.plot(x, y, label='y = 2x - 2/3')      ## The line of best fit is labelled␣
 ↪as y=2x-2/3, since the optimised m and c are extremely close to these values␣
 ↪(the
plt.plot(1,1,"ro",label='(1,1)')          ## actual values at which the minimum␣
 ↪J occurs) but the m and c from the model are used to plot the line, although
plt.plot(2,4,"ro",label='(2,4)')          ## there would be no visible␣
 ↪difference.
plt.plot(3,5,"ro",label='(3,5)')
plt.ylabel('y', fontsize=14)
plt.xlabel('x', fontsize=14)
plt.axis([0,5,0,10])
plt.title('Optimised line of best fit with learning rate 0.01 and 50,000␣
 ↪iterations ', fontsize=12)
plt.legend()
plt.show()
```



[12]:
```
## If the learning rate is too high, the cost function actually only continues␣
 ↪to increase with more and more iterations.
## This is because the gradient descent 'overshoots' the minimum, jumping past␣
 ↪it and to a point (m,c) where the cost will be even furthere from the␣
 ↪minimum.
## Upon experimenting I found the threshold for the cost diverging to be around␣
 ↪0.185.
## To demonstrate this, i have used a modified version of the function above,␣
 ↪in which how frequently the cost is printed can be inputted. This is useful␣
 ↪since J
## quickly becomes too large to calculate, so in those instances I have limited␣
 ↪the number of iterations.
## I found

def model3(alpha, num_iterations,print_num):
    m = np.random.uniform (-2,2)
    c = np.random.uniform (-2,2)
    print("J_0 = ",J(m,c))
    for iterationnumber in range(1,num_iterations+1):
        m_updated = graddescentm(m,c,alpha)
```

```
        c_updated = graddescentc(m,c,alpha)
        m = m_updated
        c = c_updated
        if iterationnumber%print_num == 0:
            print ("J_",iterationnumber,"=",J(m,c))
    return (m,c)

print("alpha = 0.18:")
print(model3(0.18,3000,500))    ## First to demonstrate the effects of a␣
 ↪learning rate that is way too large
print("alpha = 0.19:")
print(model3(0.19,3000,500))
print("alpha = 0.1:")
print(model3(0.1,10000,2000))  ## Then demonstrating that a larger learning␣
 ↪rate, when still small enough, approches the minimum quicker
print("alpha = 0.01:")
print(model3(0.01,10000,2000))

## For the sake of efficiency, when the same outcome is reached a smaller␣
 ↪number of iterations should be used. I found (alpha,num_iterations) = (0.
 ↪01,10000) to be
## suitable. Even when a smaller learning rate is used, the same value of J is␣
 ↪settled upon, as shown below.
```

```
alpha = 0.18:
J_0 =  10.01271637052363
J_ 500 = 0.5462612018090477
J_ 1000 = 0.23446258530716096
J_ 1500 = 0.22268459402898105
J_ 2000 = 0.2222396880184713
J_ 2500 = 0.2222228819813902
J_ 3000 = 0.2222222471441976
(1.999938642277661, -0.6666936580340352)
alpha = 0.19:
J_0 =  54.635554937763324
J_ 500 = 1.3715378561587457e+46
J_ 1000 = 3.4925800300602305e+90
J_ 1500 = 8.893750334050961e+134
J_ 2000 = 2.2647668578425515e+179
J_ 2500 = 5.767160902577033e+223
J_ 3000 = 1.4685902330757874e+268
(-4.710080267029027e+133, -2.0719723935729714e+133)
alpha = 0.1:
J_0 =  20.934437449360356
J_ 2000 = 0.22222222222222143
J_ 4000 = 0.22222222222222143
J_ 6000 = 0.22222222222222143
```

```
J_ 8000 = 0.22222222222222143
J_ 10000 = 0.22222222222222143
(1.9999999999999971, -0.6666666666666604)
alpha = 0.01:
J_0 =  13.493465650471729
J_ 2000 = 0.22222363771971013
J_ 4000 = 0.2222222223155086
J_ 6000 = 0.2222222222222309
J_ 8000 = 0.22222222222222143
J_ 10000 = 0.2222222222222238
(1.9999999999939977, -0.6666666666530225)
```

1.1.3 Task A3 - Machine Learning for binary classification using Logistic Regression (worth approximately 10% of the marks)

Here you will start by running some code provided to you which will create a dataset. You will use this to train a Logistic Regression classification model.

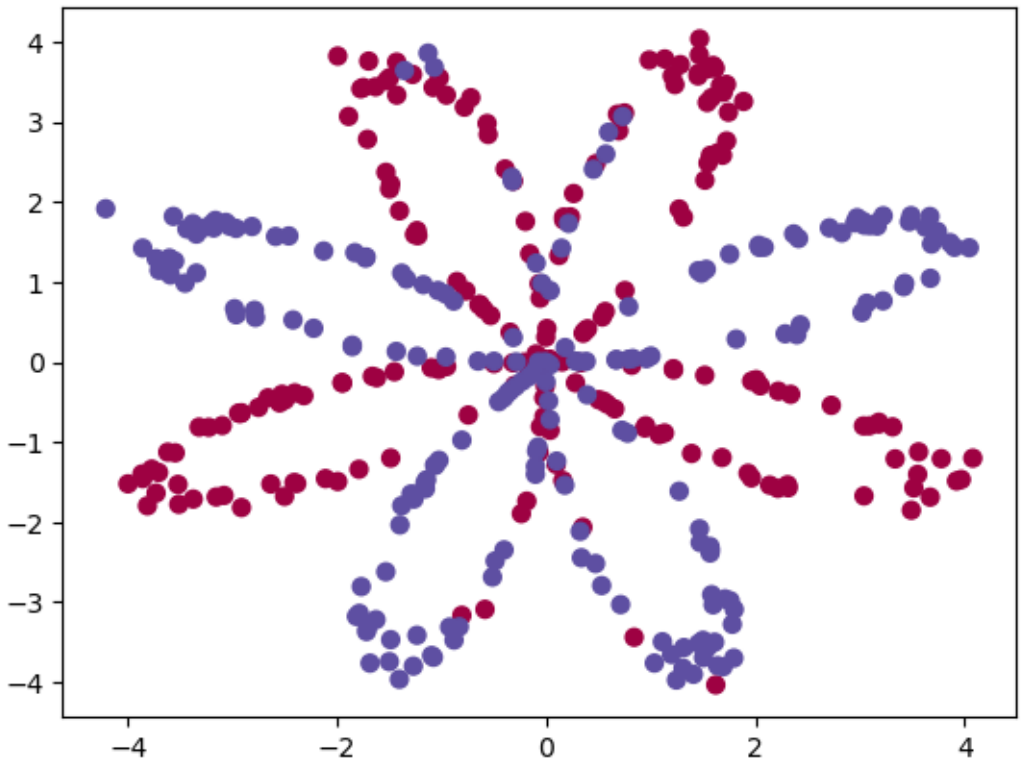
The dataset consists of a set of points (x_0, x_1) in the plane each of which is either blue or red. Your model will take as input the coordinates of a point in the plane and return 0 if the point is likely to be red, or 1 if the point is likely to be blue. In practice your model will output a float (real number) between 0 and 1 which is the probability for the point to be blue, and you will then use 0.5 as a threshold to predict the point as being blue or red.

You are guided through this task below. You will see code cells that you need to run and code cells that you need to edit/complete. You can also add new code/markdown cells as required. You may wish to add markdown cells to comment on your code or to comment on any outputs you see.

Create and visualise the dataset Run the cell below to create a planar dataset. A visualisation of this is provided, you will see that it looks like a flower with some red points and some blue points.

[13]:
```
# Create the flower dataset

import numpy as np
import matplotlib.pyplot as plt

def load_flower_dataset():
    np.random.seed(1)

    m = 400 # number of examples
    N = int(m/2) # number of points per class
    D = 2 # dimensionality

    X = np.zeros((m,D)) # data matrix where each row is a single example
    y = np.zeros((m,1), dtype='uint8') # labels vector (0 for red, 1 for blue)
    a = 4 # maximum ray of the flower
```

```
    for j in range(2):
        ix = range(N*j,N*(j+1))
        t = np.linspace(j*3.12,(j+1)*3.12,N) + np.random.randn(N)*0.2 # theta
        r = a*np.sin(4*t) + np.random.randn(N)*0.2 # radius
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        y[ix] = j

    X = X.T
    y = y.T

    return X, y

# Load the dataset:
X, y = load_flower_dataset()

# Visualize the dataset:
plt.scatter(X[0, :], X[1, :], c=y, s=40, cmap=plt.cm.Spectral);
```



You will see that you have some red points (which have label $y = 0$) and some blue points (which have label $y = 1$).

More precisely you have created two vectors `X` and `y` (shorthands for X_{train} and y_{train}) containing 400 training examples: - The numpy-array `X` of dimensions (2, 400), every column of `X` is the

feature vector for a single training example and contains the coordinates of the point: $\begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$. - The numpy-array `y` of dimensions (1,400), every column of `y` contains the label for a single training example (0 for red, 1 for blue). - The columns of `X` correspond to the columns of `y`, i.e. the point in the first column of `X` corresponds to the first entry in `y` and so on.

Your goal is to build a model to fit this data. In other words, you want the model to define regions of the plane as either red or blue. Red regions will correspond to points (x_0, x_1) where the model returns a value less than 0.5 and with any other parts of the plane being blue regions.

Let's start by building a logistic regression model and then measuring its performance on this problem. You will build it in numpy from scratch.

These are the steps to build your model.

1. Define the structure:

$$\widehat{y} = \sigma\left(W \cdot x + b\right) = \sigma\left(\begin{pmatrix} w_0 & w_1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} + b\right).$$

   Here the notation is the same as the one in the lecture presentation for this project. In this case, for each example, we take a linear combination $w_0x_0 + w_1x_1 + b$ of the input features x_0 and x_1), where w_0, w_1, b are real numbers, followed by a *sigmoid* activation function $\sigma(\lambda) = \dfrac{1}{1+\exp(-\lambda)}$. The output $\widehat{y}$ is the model's predicted probability that the point $x = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$ is blue. Here $W = \begin{pmatrix} w_0 & w_1 \end{pmatrix}$ is known as the weight matrix and b is the bias vector (which in this special case will actually be a single value, a 1×1 matrix).

2. Initialize the model's trainable parameters W and b.

3. The training loop is then as follows:

   - Implement forward propagation: compute an array whose entries are $\widehat{y}^{(i)}$ for each training example, with the current values of W and b. Here $\widehat{y}^{(i)}$ is the output probability for example in column i of X.
   - Compute the cost J for the current values of the parameters.
   - Implement backward propagation to get the gradients, i.e. the rate of change of the J with respect to each of w_0, w_1 and b at their current values.
   - Update the parameters using these gradients. This is known as gradient descent.

In practice, you'll build helper functions for each of these steps, and then merge them into one function called `model()`. Once you've built `model()` and used it to optimise the parameters, you can make predictions on new data.

Initialize parameters Complete the function `initialize_parameters` started below. It takes in the dimension of the input layer `n_X` and the dimension of the output layer `n_y` and returns returns a Python dictionary `parameters`. The dictionary `parameters` contains the initialized parameters of the model: the weight matrix `W` and the bias vector `b`.

Note that `n_X = 2` in our example, each point has two input coordinates, and the dimension of the output layer is `n_y = 1`. This means that `W` will be a 1×2 matrix and `b` will be a 1×1 matrix for our example. However we would like our function to work more generally, and be in a form which

can be adapted for use later on in this project, so we'd like the user to be able to specify `n_X` and `n_y` as inputs to `initialize_parameters`.

- Use: `np.random.randn(n,m) * 0.01` to randomly initialize a matrix of the required shape for `W`.
- Initialize the bias vector `b` so that all of its entries are zero.
- Return both of these in a dictionary called `parameters` with names W and b respectively.

Once you have completed the code run this cell.

[14]:
```
def initialize_parameters(n_X, n_y):

    #Initialize weight matrix nd bias vector
    W = W = np.random.randn(n_y,n_X) * 0.01
    b = b = np.zeros(n_y)

    #Store W and b in a dictionary parameters
    parameters = {"W": W,
                  "b": b}

    return parameters
```

Forward Propagation Next you will implement forward propagation for logistic regression:

$$z = WX + b \text{ and then } \widehat{y} = \sigma(z)$$

As above, for $\lambda \in \mathbb{R}$, $\sigma(\lambda) = \dfrac{1}{1+\exp(-\lambda)}$ (this is applied entry-wise to z in the above).

Start by implementing the sigmoid function in numpy by replacing *None* in the code cell below with appropriate code. Then run the cell.

[15]:
```
def sigmoid(z):
    sig = 1/(1 + np.exp(-1*z))
    return sig
```

Complete the function `forward_propagation` started below. It calculates the output of the logistic regression model, called `y_hat`, and returns (as a tuple) `y_hat` and `z` (in the notation given above).

Once you have completed the code run this cell.

[16]:
```
def forward_propagation(X, parameters):

    # retrieve W and b from the dictionary "parameters"
    W = parameters["W"]
    b = parameters["b"]

    # compute z
    z = np.matmul(W,X) + b
```

```
    # Compute y_hat
    y_hat = sigmoid(z)
    return y_hat, z
```

Compute cost Given the predictions $\hat{y}^{(i)}$ on all the training examples, you can compute the cost J as the average over all training examples of the losses:

$$J = \frac{1}{m}\sum_{i=0}^{m} L(\hat{y}^{(i)}, y^{(i)}).$$

Here we are using the binary cross-entropy loss:

$$L(\hat{y}^{(i)}, y^{(i)}) = -\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{y}^{(i)}\right)\right).$$

Recall $y^{(i)}$ is the i^{th} component in y, i.e. the label for example i and $\hat{y}^{(i)}$ is the models current prediction of this label, the i^{th} component in $\hat{y}$.

Complete the function `compute_cost()`, started below, to compute the value of the cost J:

1. First compute the vector of losses element-wise using the numpy arrays given as inputs `y_hat` and `y`.
2. Sum the losses and divide by the number of training examples `m`. Call this value `cost`.
3. Cast `cost` as type `float` using `float()` and return it.

Once you have completed the code run this cell.

[17]:
```
def compute_cost(y_hat, y):

    # retrieve number of training examples from the shape of y
    m = (np.shape(y))[1]

    # compute the vector of losses by computing the cross-entropy loss␣
↪element-wise
    losses = -1*(np.multiply(y,np.log(y_hat)) + np.multiply((1-y),np.
↪log(1-y_hat)))

    # compute the total cost by averaging the loss over all training examples
    cost = (1/m)*np.sum(losses)

    # cast cost as a float
    cost = float(cost)

    return cost
```

Backpropagation You can now implement backward propagation. Note that you will complete the code cell below this, much of the code is already given to you but you need to stick with the notation established.

NOTATION: in general for a generic numpy array `M` and a function F with the entries of `M` as inputs, denote by `dM` the numpy array representing the gradient of F with respect to `M`. More precisely `dM` contains the partial derivatives of F with respect to the entries of `M`. For example if

$$M = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{pmatrix}$$

and F is a function of $a_{0,0}, a_{0,1}, a_{0,2}, a_{1,0}, a_{1,1}, a_{1,2}$ then

$$dM = \begin{pmatrix} \frac{\partial F}{\partial a_{0,0}} & \frac{\partial F}{\partial a_{0,1}} & \frac{\partial F}{\partial a_{0,2}} \\ \frac{\partial F}{\partial a_{1,0}} & \frac{\partial F}{\partial a_{1,1}} & \frac{\partial F}{\partial a_{1,2}} \end{pmatrix}.$$

Throught this project F will be the cost function J.

Complete the function `backward_propagation()` started below. It takes in a dictionary `parameters` containing the current parameters, `y_hat`,`X` and `y` and it returns a dictionary containing the gradients `dW` and `db` of the cost function with respect to the trainable parameters `W` and `b`.

You are given the key code for calculating `dW` and `db`, it is, in the notation already established.

```
dz = y_hat - y
dW = np.dot(dz, X.T)/m
db = np.sum(dz, axis=1, keepdims=True)/m
```

You will see these lines of code in the cell below.

Once you have completed the code run this cell.

[18]:
```
def backward_propagation(parameters, y_hat, X, y):

    # Retrieve the number of training examples from the shape of y
    m = (np.shape(y))[1]

    # retrieve W from the dictionary "parameters".  W is not used in this␣
↪function, but I have left this line in because it was prompted.
    W = parameters["W"]

    # Backward propagation: calculate dW, and db.
    dz = y_hat - y
    dW = np.dot(dz, X.T)/m
    db = np.sum(dz, axis=1, keepdims=True)/m

    grads = {"dW": dW,
             "db": db}

    return grads
```

Update parameters Next you will define a function to update parameters using gradient descent. In a similar way to that seen in task A2,

1. W will be updated to

$$W - dW \times \text{learning rate}.$$

and

2. b will be updated to

$$b - db \times \text{learning rate}.$$

Complete the function `update_parameters()` started below. It takes in the dictionary `parameters` containing the current parameters `W` and `b`, the dictionary `grads` containing the current gradient `dW`,`dB` and the learning rate (which defaults to the value 1.2). **Once you have completed the code run this cell.**

[19]:
```
def update_parameters(parameters, grads, learning_rate = 1.2):

    # Retrieve a copy of each parameter from the dictionary "parameters"
    W = copy.deepcopy(parameters["W"])
    b = parameters["b"]

    # Retrieve each gradient from the dictionary "grads"
    dW = grads["dW"]
    db = grads["db"]

    # Update each parameter
    Wnew = W - learning_rate*dW
    bnew = b - learning_rate*db

    # Store the new parameters in the dictionary parameters
    parameters = { "W" : Wnew,
                   "b" : bnew
                 }

    return parameters
```

Model Now you will put everything together into a function `model` where you will apply `update_parameters` iteratively to update the parameters a number of times.

Complete the function `model` started below. It takes as inputs the training data `X`, `y`, a positive integer `num_iterations` which is the number of times the parameters will be updated (defaulting to $10,000$) and a boolean `print_cost` which, when set to `True`, will mean the function prints out some costs as the values of the parameters are updated). `model` return a dictionary containing the optimised parameters at the end of this process. **Once you have completed the code run this cell.**

[20]:
```
def model(X, y, num_iterations = 10000, print_cost=False):
    # Optional: to control the random seed of the initialization uncomment next␣
 ↪line:
    np.random.seed(3)

    # retrieve n_x and n_y from X and y
    n_X = np.shape(X)[0]
    n_y = np.shape(y)[0]
    # Initialize parameters
    parameters = initialize_parameters(n_X,n_y)
    # Training loop (gradient descent)
    for i in range(1, num_iterations+1):                    ## starting from 1␣
 ↪so that the indexing in the printed statement is correct

        # Forward propagation
        y_hat = (forward_propagation(X, parameters))[0]
        # Compute the cost
        cost = compute_cost(y_hat, y)
        # Backpropagation
        grads = backward_propagation(parameters, y_hat, X, y)
        # Update parameters (use learning_rate = 1.2)
        parameters = update_parameters(parameters, grads, learning_rate = 1.2)
        # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    learned_parameters = parameters

    # return learned parameters
    return learned_parameters
```

Predict on new data Below you are given a complete function `predict` that uses the learned parameters (output of `model`) and the `forward_propagation` function to predict a class for each example in the input matrix `X`. **Run the code cell below.**

[21]:
```
def predict(learned_parameters, X):

    # Compute predicted probabilities using the learned parameters
    y_hat, z = forward_propagation(X, learned_parameters)

    # Classify as 0 or 1 using 0.5 as a threshold
    predictions = (y_hat > 0.5)

    return predictions
```

Plot the results and calculate accuracy You are given the complete function `plot_decision_boundary` below which plots the decision boundary of the trained model along

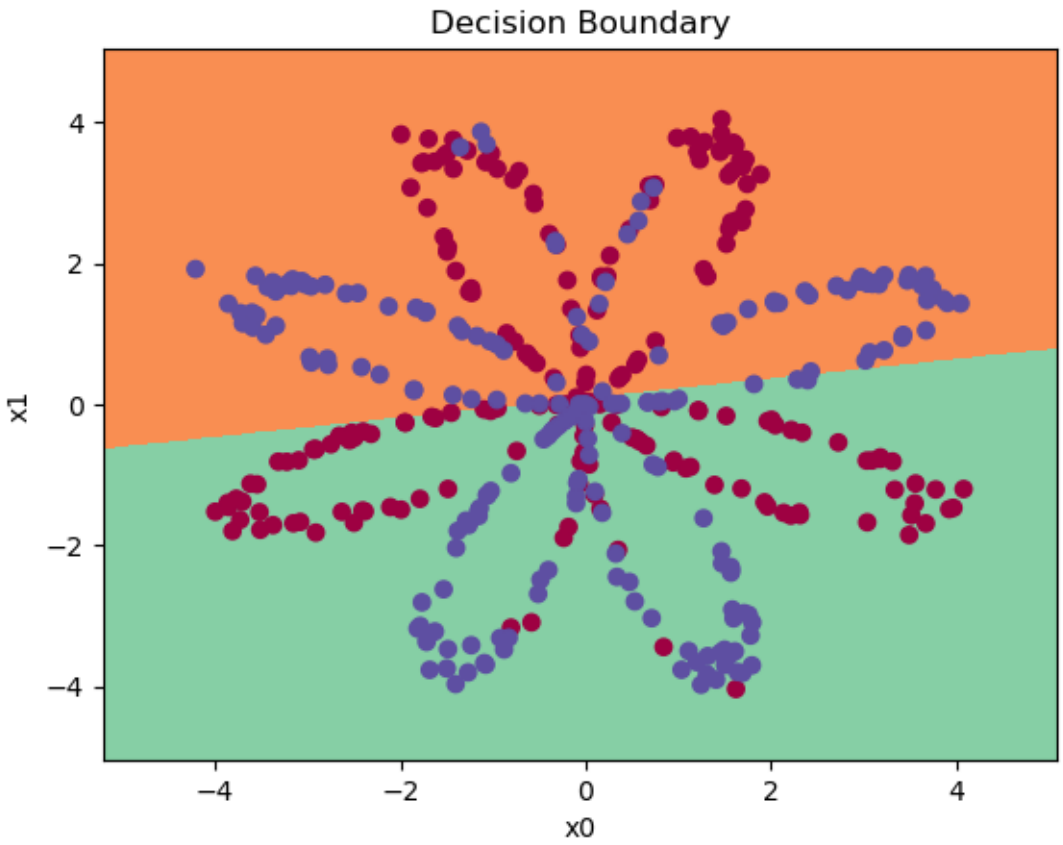
with the original data. It takes as arguments the `learned_parameters` (as output by the `model` function), the model, the training examples `X` and the true labels `y`. **Run the code cell below.**

[22]:
```
# Function to plot decision boundary

def plot_decision_boundary(learned_parameters, X, y):

    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01

    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Predict the function value for the whole grid
    Z = predict(learned_parameters,np.c_[xx.ravel(), yy.ravel()].T)
    Z = Z.reshape(xx.shape)

    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.ylabel('x1')
    plt.xlabel('x0')
    plt.scatter(X[0, :], X[1, :], c=y, cmap=plt.cm.Spectral)
    plt.title("Decision Boundary")
    plt.show()
```

Now you can train your model using the training examples, and plot the decision boundary for the learned parameters, alongside the orginal data. **Run the code cell below**.

[23]:
```
# Train your model
learned_parameters = model(X, y, num_iterations = 10000, print_cost=True)

# Plot the decision boundary
plot_decision_boundary(learned_parameters, X, y)
```

```
Cost after iteration 1000: 0.673146
Cost after iteration 2000: 0.673146

Cost after iteration 3000: 0.673146
Cost after iteration 4000: 0.673146
Cost after iteration 5000: 0.673146
Cost after iteration 6000: 0.673146
Cost after iteration 7000: 0.673146
Cost after iteration 8000: 0.673146
Cost after iteration 9000: 0.673146
Cost after iteration 10000: 0.673146
```



As you should see logistic regression doesn't do a good job in predicting the labels for this dataset. This is because the dataset is not linearly separable, and therefore a simple logistic regression performs poorly on this data.

Complete the code cell below with a formula for the accuracy of the model. This should be the precentage of points for which the model gives the correct classification. **Then run the cell.**

[24]:
```
# Print accuracy

predictions = predict(learned_parameters, X)

total = 0
for i in range (0,np.shape(X)[1]):
    if y[:,i] == 1 and predict(learned_parameters, X)[:,i]:
        total +=1
    else:
        if y[:,i] == 0 and predict(learned_parameters, X)[:,i] == False:
           total +=1

accuracy = 100*(1/(np.shape(X)[1]))*total
```

```
print ('Accuracy of logistic regression: %d ' % accuracy +
       '% ' + "(percentage of correctly labelled datapoints)")
```

```
Accuracy of logistic regression: 47 % (percentage of correctly labelled
datapoints)
```

1.1.4 Task A4 - Build a hidden layer Neural Network (worth approximately 20% of the marks)

You are now going to build a full Neural Network with one hidden layer. This will allow to model to learn more complex decision boundaries and not just linear ones. This will result in an improved model compared to that you obtained in task A3.

In practice you are going to repeat all the steps of task A3 but this time with an extra layer to the network. Set the hidden layer to have 4 neurons (in the code this dimension is denoted `n_h`). You may find it helpful to refer to the project lecture when you are completing this section.

The forward propagation has the form `linear->tanh->linear->sigmoid`. So this will take the form below where W and T are appropriately sized matrices containing weights and b and c are appropriately sized column vectors for the biases (some of the dimensions of these objects will now depend on `n_h`). The input matrix X is the same as in task A3, z, a, v give intermediate output with $\hat{y}$ the final output.

$$z = W \cdot X + ba = \tanh(z)v = T \cdot a + c\hat{y} = \sigma(v)$$

The code for the new backward propogation function, in this notation, is as follows and you can cut and paste this into your code as appropriate.

```
dv = y_hat - y
dT = np.dot(dv, a.T)/m
dc = np.sum(dv, axis=1, keepdims=True)/m
dz = np.dot(T.T, dv) * (1 - np.power(a, 2))
dW = np.dot(dz, X.T)/m
db = np.sum(dz, axis=1, keepdims=True)/m
```

The cost is the same as for logistic regression, so there is no need to implement it again.

Neural Network Model Below gives a basic structure for your final model, you need to build all the helper functions from scratch this time.

Specifically, using an approach similar to that seen in A3, you should

- Complete the function below `model_nn`, including all the necessary helper functions.
- Include a function `predict` which takes in the dictionary of paremeters and input data 'X' and returns predictions of the labels for each example.
- Produce a visualisation of the regions of the plane predicted to be red/blue by your model and the original data.
- Calculate and print the accuracy of the model (the percentage of points for which the model predicts the correct label).

- Make some comments, in a markdown cell, which compare the model developed in A3 to the model developed in A4.

You may use code from Task A3 here but be careful to rename functions and variables as appropriate to avoid clashes with any values established for this notebook in Task A3.

Edit the code cell below and insert further code and markdown cells in which to answer this task

[25]:
```
def A4_initialize_parameters(n_X, n_y, n_h = 4):  #Initializing parameters␣
↪W,b,T,c from input (# of inputs, # of outputs, # of hidden layer neurons)
   A4_W = np.random.randn(n_h,n_X)  #W is a 4,2 matrix with random values from␣
↪a normal distribution
   A4_b = np.zeros(n_h).reshape(n_h,n_y)  #b is a 4,1 zero matrix
   T = np.random.randn(n_y,n_h) #T is a 1,4 matrix with random values from a␣
↪normal distribution
   c = 0   #c is a scalar 0
   A4_parameters = {"A4_W": A4_W,
                    "A4_b": A4_b,
                    "T": T,
                    "c": c}
   return A4_parameters          #Function returns library containing W,b,T,c
```

[26]:
```
def A4_forward_propagation(X, A4_parameters):     #Computing yhat through the␣
↪two activation functions
   A4_W = A4_parameters["A4_W"]                   #Retrieving parameters from␣
↪the library
   A4_b = A4_parameters["A4_b"]
   T = A4_parameters["T"]
   c = A4_parameters["c"]

   A4_z = np.matmul(A4_W,X) + A4_b                #z = W.X + b
   a = np.tanh(A4_z)                              #a = tanh(z)
   v = np.matmul(T,a) + c                         #v = T.a + c
   A4_y_hat = sigmoid(v)                          #yhat = sig(v)
   return A4_y_hat, a                             #Returns both yhat and a (need␣
↪a too because it's used in the differentiation for  dT and dz)
```

[27]:
```
def A4_backward_propagation(A4_parameters, A4_y_hat, a, X, y):    #Function to␣
↪calculate derivatives, input: (parameters, yhat, a, input, training output)
   m = (np.shape(y))[1]                                          #m = number␣
↪of points = 400
   T = A4_parameters["T"]                                        #Retrieving T␣
↪from library (to be used in line dz=...)

   dv = A4_y_hat - y                                             #Calculating␣
↪derivatives of v,T,c,z,W,b  (code given by project specification)
   dT = np.dot(dv, a.T)/m
```

```
    dc = np.sum(dv, axis=1, keepdims=True)/m
    dz = np.dot(T.T, dv) * (1 - np.power(a, 2))
    A4_dW = np.dot(dz, X.T)/m
    A4_db = np.sum(dz, axis=1, keepdims=True)/m
    A4_grads = {"A4_dW": A4_dW,                                    #Puts␣
↪derivatives of the 4 parameters in a library
             "A4_db": A4_db,
             "dT": dT,
             "dc": dc}
    return A4_grads                                                #Output:␣
↪Library with dW,db,dT,dc
```

[28]:
```
def A4_update_parameters(A4_parameters, A4_grads, learning_rate = 1.2):  ␣
↪#Updating parameters by taking a step in direction of steepest descent,␣
↪input: parameters library,grads library, learning rate
    A4_W = A4_parameters["A4_W"]                              #Retrieving the 4␣
↪parameters from the library
    A4_b = A4_parameters["A4_b"]
    T = A4_parameters["T"]
    c = A4_parameters["c"]

    A4_dW = A4_grads["A4_dW"]                                                ␣
↪#Retrieving the 4 derivatives from the library
    A4_db = A4_grads["A4_db"]
    dT = A4_grads["dT"]
    dc = A4_grads["dc"]

    A4_Wnew = A4_W - learning_rate*A4_dW                                     ␣
↪#Calculating: new parameter = old parameter - (learning rate*derivative)
    A4_bnew = A4_b - learning_rate*A4_db
    Tnew = T - learning_rate*dT
    cnew = c - learning_rate*dc

    A4_parameters = { "A4_W" : A4_Wnew,                                  #Puts␣
↪new parameters into the parameters library
                      "A4_b" : A4_bnew,
                      "T" : Tnew,
                      "c" : cnew}
    return A4_parameters                                                     ␣
↪#Output: library with updated parameters
```

[29]:
```
def model_nn(X, y, num_iterations = 10000, print_cost=False):            ␣
↪#Model Input: Training features, Training labels, number of iterations

    # Optional: to control the random seed of the initialization uncomment next␣
↪line:
```

```
    #np.random.seed(3)

    # retrieve n_x and n_y from X and y
    A4_n_X = np.shape(X)[0]
    A4_n_y = np.shape(y)[0]
    # Initialize parameters
    A4_parameters = A4_initialize_parameters(A4_n_X,A4_n_y)

    # Training loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation
        A4_y_hat, a = A4_forward_propagation(X, A4_parameters)
        # Compute the cost
        A4_cost = compute_cost(A4_y_hat, y)
        # Backpropagation
        A4_grads = A4_backward_propagation(A4_parameters, A4_y_hat, a, X, y)
        # Update parameters (use learning_rate = 1.2)
        A4_parameters = A4_update_parameters(A4_parameters, A4_grads,␣
↪learning_rate = 1.2)
        # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, A4_cost))

    A4_learned_parameters = A4_parameters

    # return learned parameters
    return A4_learned_parameters
```

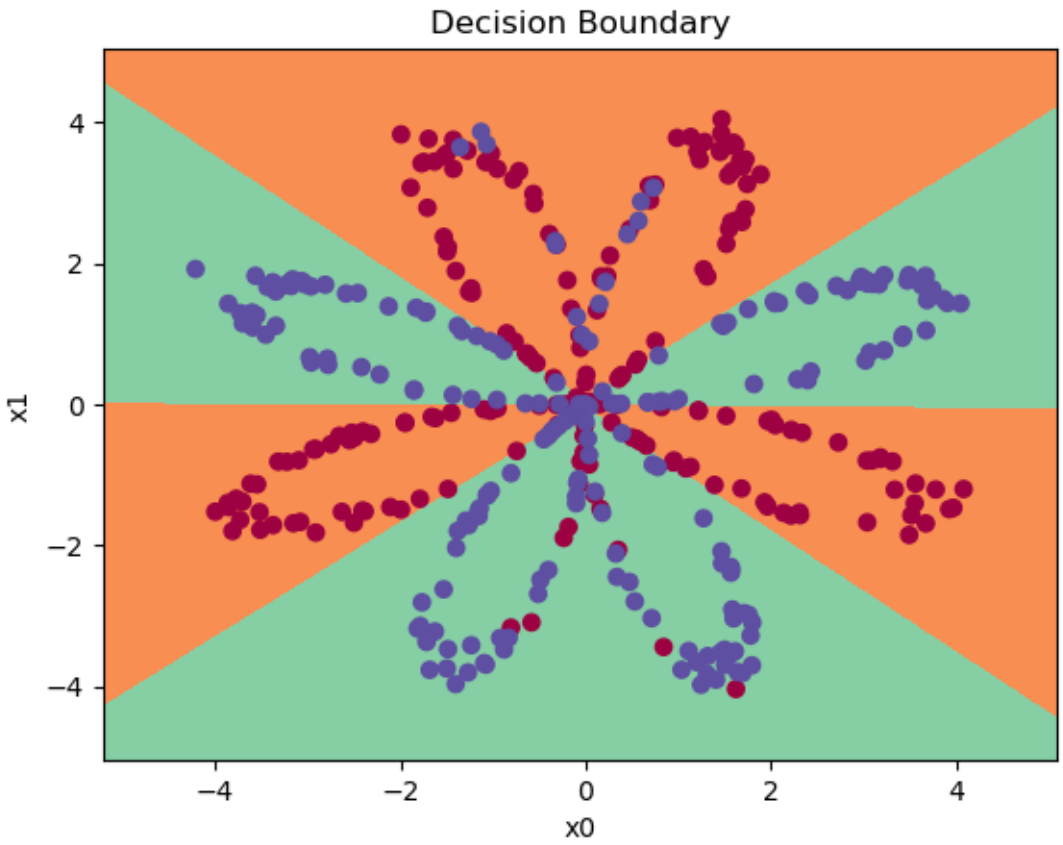
[30]:
```
def A4_predict(A4_learned_parameters, X):
    # Compute predicted probabilities using the learned parameters

    A4_y_hat = A4_forward_propagation(X, A4_learned_parameters)[0]

    # Classify as 0 or 1 using 0.5 as a threshold

    A4_predictions = (A4_y_hat > 0.5)                    #Returns True if yhat>0.
↪5 so True if model believes point is blue

    return A4_predictions                                #list of True or False␣
↪for every point
```

[31]:
```
# Function to plot decision boundary
def A4_plot_decision_boundary(A4_learned_parameters, X, y):

    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
```

```
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01

    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # Predict the function value for the whole grid
    A4_Z = A4_predict(A4_learned_parameters,np.c_[xx.ravel(), yy.ravel()].T)
    A4_Z = A4_Z.reshape(xx.shape)

    # Plot the contour and training examples
    plt.contourf(xx, yy, A4_Z, cmap=plt.cm.Spectral)
    plt.ylabel('x1')
    plt.xlabel('x0')
    plt.scatter(X[0, :], X[1, :], c=y, cmap=plt.cm.Spectral)
    plt.title("Decision Boundary")
    plt.show()
```

[32]:
```
# Train your model
A4_learned_parameters = model_nn(X, y, num_iterations = 10000, print_cost=True)

# Plot the decision boundary
A4_plot_decision_boundary(A4_learned_parameters, X, y)
```

```
Cost after iteration 0: 0.705839
Cost after iteration 1000: 0.286515

Cost after iteration 2000: 0.273709

Cost after iteration 3000: 0.266685
Cost after iteration 4000: 0.262247

Cost after iteration 5000: 0.259201

Cost after iteration 6000: 0.256994
Cost after iteration 7000: 0.255330

Cost after iteration 8000: 0.254033

Cost after iteration 9000: 0.252995
```



[33]:
```
# Print accuracy
A4_predictions = A4_predict(A4_learned_parameters, X)
A4_total = 0                                                    #Start with␣
 ↪total correct points = 0
for i in range (0,np.shape(X)[1]):
    if y[:,i] == 1 and A4_predictions[:,i]:                     #If point is␣
 ↪blue and model predicts it's blue then add 1 to total correct points
        A4_total +=1
    else:
        if y[:,i] == 0 and A4_predictions[:,i] == False:        #If point is␣
 ↪red and model predicts it's red then add 1 to total correct points
            A4_total +=1


accuracy = 100*(1/(np.shape(X)[1]))*A4_total                    #Accuracy % is␣
 ↪100*(total correct point/total points)

print ('Accuracy of logistic regression: %d ' % accuracy +
       '% ' + "(percentage of correctly labelled datapoints)")
```

```
Accuracy of logistic regression: 91 % (percentage of correctly labelled
datapoints)
```

The Model with the hidden layer made in A4 is much more accurate than the simple one in A3. It's ability to split the plane in a more complex way than just a linear split allows it to fit the flower shape more effectively. The A4 model has an accuracy of approximately 90% while the A3 model is around 47% accurate. Not only is it correct for a larger number of the points in the sample, it is also more certain in the predictions. If we change the 0.5 in the predict function, (i.e change the threshold an output must reach to be predicted blue), the line between blue and orange in A3 shifts with the change, meaning that there is a gradient of certainty with a lot of uncertainty near the border. On the other hand, the predictions diagram for the model in A4 doesn't change noticeably unless the threshold is moved below 0.15 or above 0.7, and even then only some sections change. In other words, the model in A4 has more concrete, confident blue and red sections, whereas A3 gives a gradient from blue at the bottom to orange at the top.

1.2 Section B (worth approximately 40%)

Remember you are free to use sklearn in this section.

1.2.1 Task B1

- How can we further improve neural networks? You may wish to talk about underfitting, overfitting, regularisation and use examples to illustrate you ideas.

1.2.2 Task B2

- What are some other machine learning models (e.g. decision trees). Give some examples. What are improvements to these (e.g random forest, xgboost in the case of decision trees)?

You may wish to use datasets from https://archive.ics.uci.edu/ in this section.

1.3 Overfitting

In general, one prefers a model that has less variance with some training error over a model with high variance but a very low training error. This is because a model that has high variance and fits the data exactly cannot generalise as it hasn't understood the pattern but has just learnt the data. This is useless for practical application on new data. In this situation the model is said to be overfitting and there are a number of techniques to prevent this such as L1 and L2 regularization, early stopping and drop-out regularisation.

Complex Neural Networks without regularisation are prone to overfitting as they have a lot of parameters and thus are more equipped to learn the data perfectly by learning the noise within the training data. To solve this, we want to consider the sensitivity of the model to the inputs, something which is determined by the neural network's structure and the weights.

1.3.1 General Regularization Approach

Often, an approach is to add a regularisation term to the loss function in order to penalise the model for high weights. In a general form this looks like the following expression,

$$L_R = L\left(\hat{y}^{(i)}, y^{(i)}\right) + \lambda R(\mathbf{w}) \tag{1}$$

Where $\mathbf{w}$ is the weight vector, λ the regularisation strength, $L\left(\hat{y}^{(i)}, y^{(i)}\right)$ the original loss function (for example, binary cross-entropy loss as used in section A) and R a regularization term. λ is a hyperparameter that one chooses to determine how strong the regularisation is.

What this technique aims to do is as the model tries to minimise the regularised loss function, it will minimise the original loss function, $L\left(\hat{y}^{(i)}, y^{(i)}\right)$, and the regularisation term, $R(\mathbf{w})$, which will yield a less complex model. If the weights are large, the regularisation term is designed to penalise the overall loss function and so will encourage the weight vector to become smaller. This will decrease the variance while also training the model to still make good predictions.

1.3.2 L1 Regularisation

L1 regularisation follows this approach and is characterised by its regularisation term, the absolute value of the weights.

$$L_R = L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{n}\|\mathbf{w}\|_1$$

Where $\|\mathbf{w}\|_1 = \sum_{i=1}^{N} |w_i|$. This is known as the L1 norm.

As above, this term is a penalty against the complexity of the model as a positive value is always added. Large weights make the model more sensitive to small changes in the inputs of the model and so what the L1 norm does is encourage those features which are less important to disappear.

By reducing the number of irrelevant independent variables by its tendency to reduce the weights of such variables to zero, this penalty term yields what is called a 'sparse' model. Thus, L1 makes the model robust to outliers and noise. This penalty term achieves this because of the linearity of the absolute value term which forms sharp intersections with zeros of the weights (as in the figure below). This means this with large enough λ, it can make some weights exactly zero.

1.3.3 L2 Regularisation

L2 Regularisation is of the general form (1), like L1, but the regularisation term is instead the sum of the squared values of the weights.

$$L_R = L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{n}\|\mathbf{w}\|_2^2$$

Where $\|\mathbf{w}\|_2^2 = \sum_{i=1}^{N} w_i^2$, known as the L2 Norm.

This regularised loss function works similarly to L2, however the circular property of the L2 Norm means it reduces the weights more gradually than L1, so they tend towards zero but are less likely to become exactly zero. This means that every weight still makes a contribution to the model rather than creating a sparse model.

We can see this illustrated in the figure below which is a geometric representation of how regularisation affects the minimisation of the loss function. The plots are in 2-D space, and the axes represent two distinct weights factors. The elliptic contours is the surface plot of the loss function

(for example the mean squared error for linear regression). Each curve represents the graph of the points which give a specific magnitude of the loss function, and the point $\hat{\beta}$ is the minimum point of the loss function.

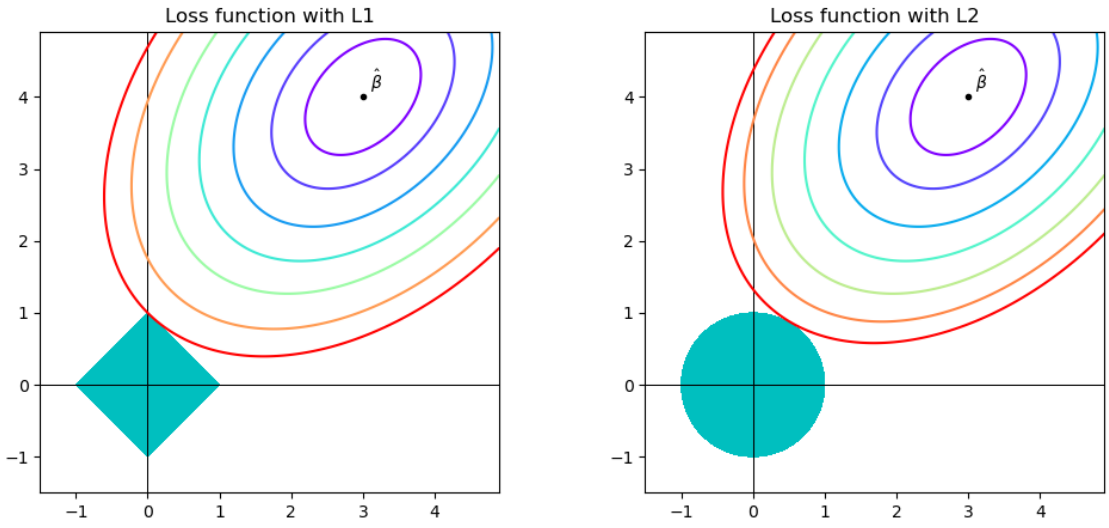
The figure on the left features a plot of the L1 norm, which is the diamond shape constraint curve. The right figure shows the L2 norm, a circle. These are added to the loss function respectively. For both, as you move away from the minimum norm value, $(0, 0)$, the magnitudes of the weights increase.

As we have seen, we want to minimise both the weights and the loss function. This combination of loss function and the regularisation will be minimised at some point where they touch each other, as we do not want to have such small weights that the magnitude of the loss function is too large.

Notice how for the L1 case, the solution is where one of the weight factors is zero and for L2 both weights are nonzero.

[34]:
```
import numpy as np
import matplotlib.pyplot as plt


def plot_l1_l2_contour_plot():
    # Plotting code taken largely from https://scikit-learn.org/stable/
 ↪auto_examples/linear_model/plot_sgd_penalties.html

    # Parameters for elliptic paraboloid loss function
    x_scale = 2
    y_scale = 3
    centre = (3, 4)
    angle = np.pi / 4

    plt_bounds = (-1.5, 4.9)

    line = np.linspace(*plt_bounds, 1001)
    xx, yy = np.meshgrid(line, line)

    # See https://www.desmos.com/3d/i3dyObzsee for my interactive Desmos graph
    loss_func = (((xx - centre[0]) * np.cos(angle) - (yy - centre[1]) * np.
 ↪sin(angle)) / x_scale)**2 + \
        (((yy - centre[1]) * np.cos(angle) + (xx - centre[0]) * np.sin(angle)) /
 ↪ y_scale)**2

    l1 = np.abs(xx) + np.abs(yy)
    l2 = xx**2 + yy**2

    fig, (l1_ax, l2_ax) = plt.subplots(
        1,
        2,
        subplot_kw={
            "aspect": 1,
```

```
            "xlim": plt_bounds,
            "ylim": plt_bounds,
        }
    )
    fig.set_size_inches(12, 5)

    for ax in (l1_ax, l2_ax):
        ax.axhline(0, color='k', lw=0.75)
        ax.axvline(0, color='k', lw=0.75)

        ax.plot(*centre, '.k')
        ax.annotate(r"$\hat{\beta}$", centre, xytext=(centre[0] + 0.1,␣
↪centre[1] + 0.1))

    l1_ax.set_title("Loss function with L1")
    l2_ax.set_title("Loss function with L2")

    l1_ax.contourf(xx, yy, l1, levels=[0, 1], colors="c")
    l2_ax.contourf(xx, yy, l2, levels=[0, 1], colors="c")

    l1_ax.contour(xx, yy, loss_func, cmap="rainbow", levels=[0.1, 0.25, 0.5, 0.
↪8, 1.15, 1.6, 2])
    l2_ax.contour(xx, yy, loss_func, cmap="rainbow", levels=[0.1, 0.25, 0.5, 0.
↪8, 1.15, 1.5, 1.8])

    plt.show()
    plt.clf()


plot_l1_l2_contour_plot()
```



```
<Figure size 640x480 with 0 Axes>
```

1.3.4 Comparing the L1 and L2 norms through their affect on gradient descent

We can better understand the different effects of the norms through their affect on the partial derivatives of the loss function with respect to the weights.

For L2, recall

$$L_R = L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{n}\|\mathbf{w}\|_2^2$$

If we take the partial derivatives with respect to the individual weight components, we obtain

$$\frac{\partial L_R}{\partial w_i} = \frac{\partial L}{\partial w_i} + \frac{\lambda}{n} w_i$$

As in A2, we can calculate $\dfrac{\partial L}{\partial w_i}$ as usual, and then we see that the regularised learning rule is,

$$w_i \mapsto w_i - \alpha\frac{\partial L}{\partial w_i} - \frac{\alpha\lambda}{n} w_i = \left(1 - \frac{\alpha\lambda}{n}\right) w_i - \alpha\frac{\partial L}{\partial w_i}$$

So the weight factor is rescaled by a factor of $1 - \dfrac{\alpha\lambda}{n}$.

For L1, the formula for the regularised loss function is

$$L_R = L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{n}\|\mathbf{w}\|_1$$

Differentiating, we obtain

$$\frac{\partial L_R}{\partial w_i} = \frac{\partial L}{\partial w_i} + \frac{\lambda}{n}\operatorname{sgn}(w_i)$$

And the resulting learning rule is,

$$w_i \mapsto w_i - \alpha\frac{\partial L}{\partial w_i} - \frac{\alpha\lambda}{n}\operatorname{sgn}(w_i)$$

L2 shrinks the weight by a value which is proportional to the weight, whereas L1 does this by taking a constant value which pushes the weights towards zero.

When $|w_i|$ is very large, L2 will reduce the weight much more than L1, but when it is smaller, L1 reduces the weight much more than L2 does.

The key difference between the two regularisation terms is that L1 results in sparser models, where only the most important features are nonzero whereas L2 reduces the magnitude of all the weights so that they are kept close to zero, but not exactly zero. For situations where we might be more interested in feature selection, we might prefer to use L1, however if we want a more stable model where all the parameters have some relevance, L2 might be preferred.

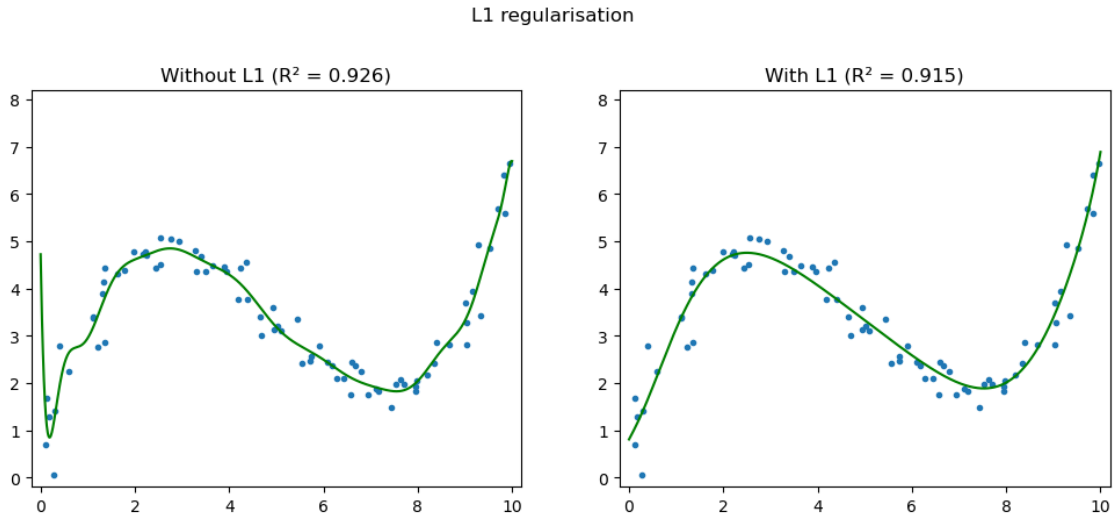
Now we will present a few examples of overfitting. We generate 80 points on a cubic curve and give each one a small nudge in a random direction, and then train a model to fit a 20 degree polynomial to the points. The high degree gives the model freedom to overfit, so we counter this with L1 and L2 regularisation.

[35]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import *
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler, PolynomialFeatures


def overfitting_line_func(x):
    return x**3 / 15 - x**2 + 4 * x


def generate_overfitting_data():
    N = 80
    max_offset = 0.5

    xs = np.linspace(0, 10, N)
    ys = np.array(overfitting_line_func(xs), dtype=xs.dtype)
    for i in range(N):
        # Give each point a small random offset
        angle = np.random.random() * 2 * np.pi
        dist = max_offset * np.random.random()
        xs[i] += dist * np.cos(angle)
        ys[i] += dist * np.sin(angle)

    return (xs, ys)


def plot_overfitting_example(
    title: str,
    left_model,
    left_label: str,
    right_model,
    right_label: str,
    *,
    left_polynomial_degree: int = 20,
    right_polynomial_degree: int = 20
) -> None:
    (xs, ys) = generate_overfitting_data()
    X_train = xs.reshape(-1, 1)
    Y_train = ys

    # Now do the neural networks
```

```
    left_pipeline = make_pipeline(
        StandardScaler(),
        PolynomialFeatures(degree=left_polynomial_degree, include_bias=False),
        left_model,
    )
    left_pipeline.fit(X_train, Y_train)

    right_pipeline = make_pipeline(
        StandardScaler(),
        PolynomialFeatures(degree=right_polynomial_degree, include_bias=False),
        right_model,
    )
    right_pipeline.fit(X_train, Y_train)

    left_score = left_pipeline.score(X_train, Y_train)
    right_score = right_pipeline.score(X_train, Y_train)

    # Now plot what the models predict
    fig, (left_ax, right_ax) = plt.subplots(
        1,
        2,
        subplot_kw={
            "aspect": 1,
            "xlim": (-0.2, 10.2),
            "ylim": (-0.2, 8.2),
        }
    )
    fig.suptitle(title)
    fig.set_size_inches(12, 5)

    left_ax.plot(xs, ys, ".")
    right_ax.plot(xs, ys, ".")

    left_ax.set_title(f"{left_label} (R² = {left_score:.3f})")
    right_ax.set_title(f"{right_label} (R² = {right_score:.3f})")

    pred_xs = np.linspace(0, 10, 1001)
    left_pred_ys = left_pipeline.predict(pred_xs.reshape(-1, 1))
    right_pred_ys = right_pipeline.predict(pred_xs.reshape(-1, 1))

    left_ax.plot(pred_xs, left_pred_ys, "-g")
    right_ax.plot(pred_xs, right_pred_ys, "-g")

    plt.show()
    plt.clf()
```

[36]:
```
plot_overfitting_example(
    "L1 regularisation",
    LinearRegression(), # Lasso with alpha=0 is just LinearRegression
    "Without L1",
    Lasso(alpha=0.01),
    "With L1",
)
```

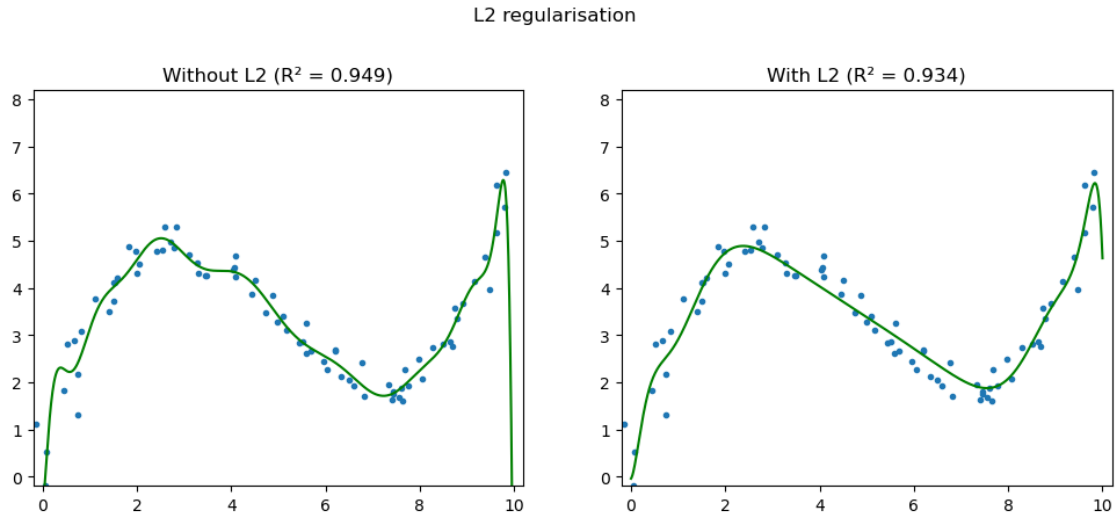
```
/nix/store/hf1jahh9nn2scy4nj34si91l4wx4f657-python3-3.12.8-
env/lib/python3.12/site-
packages/sklearn/linear_model/_coordinate_descent.py:697: ConvergenceWarning:
Objective did not converge. You might want to increase the number of iterations,
check the scale of the features or consider increasing regularisation. Duality
gap: 7.785e+00, tolerance: 1.407e-02
  model = cd_fast.enet_coordinate_descent(
```




```
<Figure size 640x480 with 0 Axes>
```

[37]:
```
plot_overfitting_example(
    "L2 regularisation",
    Ridge(alpha=0.0, max_iter=10_000),
    "Without L2",
    Ridge(alpha=1.0, max_iter=10_000),
    "With L2",
)
```

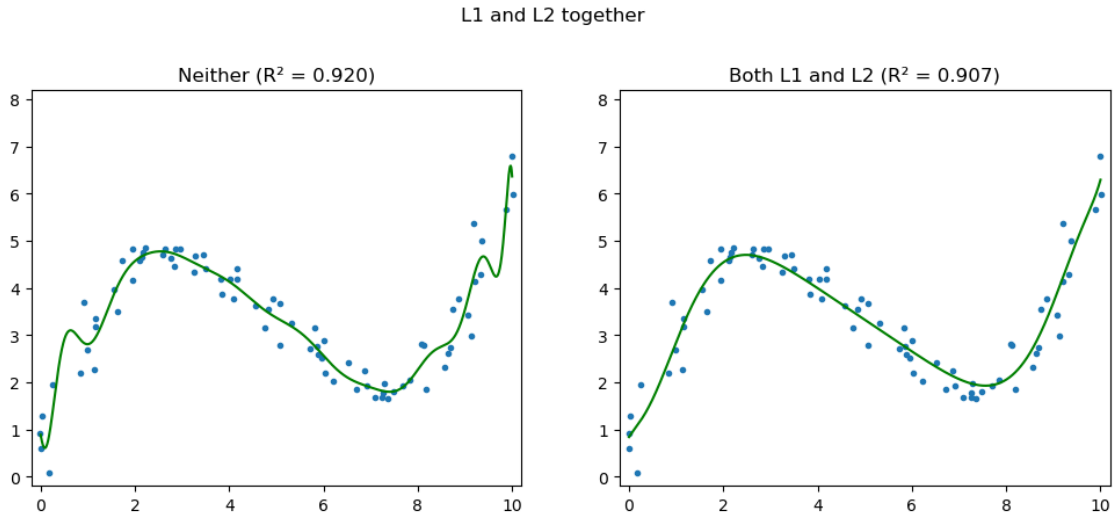
```
/nix/store/hf1jahh9nn2scy4nj34si91l4wx4f657-python3-3.12.8-
env/lib/python3.12/site-packages/sklearn/linear_model/_ridge.py:216:
LinAlgWarning: Ill-conditioned matrix (rcond=3.86577e-18): result may not be
accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
```



<Figure size 640x480 with 0 Axes>

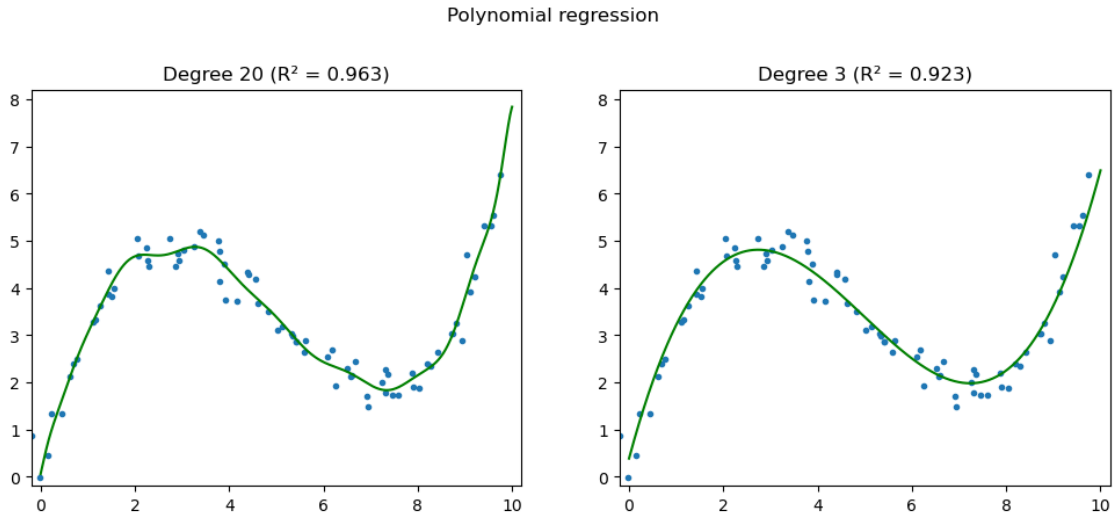
[38]:
```
plot_overfitting_example(
    "L1 and L2 together",
    LinearRegression(),
    "Neither",
    ElasticNet(alpha=0.01, max_iter=10_000),
    "Both L1 and L2",
)
```

```
/nix/store/hf1jahh9nn2scy4nj34si91l4wx4f657-python3-3.12.8-
env/lib/python3.12/site-
packages/sklearn/linear_model/_coordinate_descent.py:697: ConvergenceWarning:
Objective did not converge. You might want to increase the number of iterations,
check the scale of the features or consider increasing regularisation. Duality
gap: 7.675e+00, tolerance: 1.355e-02
  model = cd_fast.enet_coordinate_descent(
```




```
<Figure size 640x480 with 0 Axes>
```

[39]:
```
plot_overfitting_example(
    "Polynomial regression",
    LinearRegression(),
    "Degree 20",
    LinearRegression(),
    "Degree 3",
    left_polynomial_degree=20,
    right_polynomial_degree=3
)
```



```
<Figure size 640x480 with 0 Axes>
```

1.3.5 Dropout Regularisation and Early stopping

There are other techniques to combat overfitting which do not take the form of adding a regularisation term to the loss function.

Dropout regularisation is where in every training step, each non-output neuron has a probabililty p of being temporarily inactive, along with its connections. The parameter p is called the dropout rate, which is usually somewhere between 0.2 and 0.5. However, if we use a dropout rate too high, for example 0.7 or above, it can lead to under-fitting. So, we need to tune the parameter to the desired level of effectiveness.

In each training loop, the network is trained on a subnetwork with some neurons and their connections disabled. Training a neural network using dropout regularisation is training a collection of possible subnetworks, and then merging them into one. So the model learns not to be too reliant on any one hidden unit as each one can be randomly absent. This makes the model more robust to noise in the training data. Thus, not only does dropout prevent overfitting, but it also results in these more robust models. Dropout has been, and is, used in advanced and complex computer imaging models, where there are many parameters to take into account and not overfit.

Early stopping is where you follow the model's validation error after each epoch (a whole pass through the training data) and if the validation starts to increase, you stop the training process. This is done because an increase in validation error indicates that the network is memorizing the training data and becoming increasingly unable to generalise.

However, early stopping doesn't always improve effectiveness, as stopping the training process early can also mean stopping it prematurely, meaning the model hasn't learnt enough or as much as it could have to make a better model. Moreover, the point at which you stop the learning process is dependent on the validation data you select as the stopping point is as a function of the validation error. This will lead to different stopping points and thus there is a possibility of premature stopping of the learning process due to the validation data you pick. But early stopping has a very simple implementation method and is mostly effective and thus is widely used.

1.4 Notes about this submission

You will submit a single Jupyter notebook for this project assignment. Details will be provided for this on the MA124 Moodle page.

- The last thing you should do before submitting the notebook is to Restart Kernel and Run All Cells. You should then save the notebook and submit the .ipynb file. **You will lose marks if you submit a notebook that has not been run.**
- You are expected to add code and markdown cells to this document as appropriate to provide your responses to the tasks. Instructions about this are given throughout but feel free to add markdown cells at any point to provide clarity or comments.
- Likewise, to help the reader, please provide appropriate comments in your code (for example functions or blocks of code should have comments about what they do, variables should be described in comments, as appropriate).